



توضیحات فایل‌های فاز اول پروژه کامپایلر¹

تیم پروژه کامپایلر - پاییز ۱۴۰۳

2.....	Compiler.cpp یا Main.cpp
2.....	Lexer.h
5.....	Lexer.cpp
6.....	شناسایی شناسه‌ها و کلمات کلیدی
7.....	شناسایی اعداد
8.....	شناسایی علائم خاص (عملگرها و نشانه‌ها)
10.....	AST.h
10.....	خلاصه اجزا
13.....	Parser.h
13.....	Parser.cpp
16.....	Sema.h
16.....	Sema.cpp
17.....	CodeGen.cpp
18.....	CodeGen.h
18.....	روش تست هر فایل به شکل مجزا
20.....	نحوه لاگ گذاشتن در کد برای دیباگ
20.....	لینک پروژه‌های گیت‌هاب ترم قبل
21.....	لینک پروژه‌های گیت‌هاب دو ترم قبل

¹ برای اطلاعات بیشتر به فصل سوم کتاب Learn LLVM 12 مراجعه کنید.

Main.cpp یا Compiler.cpp

این فایل، فایل اصلی پروژه است که به ترتیب فایل‌های لکسر، پارسر، سمنتیک و کدجن را اجرا می‌کند.

```
// Create a lexer object and initialize it with the input expression.
Lexer Lex(Input);

// Create a parser object and initialize it with the lexer.
Parser Parser(Lex);

// Parse the input expression and generate an abstract syntax tree (AST).
Program *Tree = Parser.parse();

// Check if parsing was successful or if there were any syntax errors.
if (!Tree || Parser.hasError())
{
    llvm::errs() << "Syntax errors occurred\n";
    return 1;
}

// Perform semantic analysis on the AST.
Sema Semantic;
if (Semantic.semantic(Tree))
{
    llvm::errs() << "Semantic errors occurred\n";
    return 1;
}

// Generate code for the AST using a code generator.
CodeGen CodeGenerator;
CodeGenerator.compile(Tree);
```

Lexer.h

فایل lexer.h شامل کلاس‌های Lexer و Token است که با همکاری یکدیگر به تقسیم کد source به توکن‌ها می‌پردازند. کلاس Token نماینده توکن‌های جداگانه‌ای است که دارای نوع و متن هستند، در حالی که Lexer از طریق بافر ورودی کد را می‌خواند و بر اساس الگوهای تشخیص داده شده، توکن‌ها را تشکیل می‌دهد.

مفاهیم کلیدی:

- توکن‌ها: واحدهای بنیادی در کد منبع که بر اساس (TokenKind) دسته‌بندی می‌شوند.
- مدیریت بافر: Lexer ورودی را از یک بافر می‌خواند، با پیشروی در میان کاراکترها توکن‌ها را تشکیل می‌دهد.
- تشکیل توکن: Lexer از متد formToken() برای تکمیل توکن‌ها بر اساس آنچه از ورودی می‌خواند استفاده می‌کند.

TokenKind Enumeration

```
public:
    enum TokenKind : unsigned short
    {
        eoi,           // end of input
        unknown,       // in case of error at the lexical level
        ident,         // identifier (e.g., variable names)
        number,        // integer literal
        // Operators and other symbols follow...
    };
```

نوع‌های مختلف توکن‌ها (مانند eoi برای پایان ورودی، ident برای شناسه‌ها، number برای اعداد و ...) در اینجا تعریف شده‌اند. این توکن‌ها برای طبقه‌بندی اجزای ورودی مانند کلمات کلیدی، عملگرها و علائم استفاده می‌شوند.

```
public:
    TokenKind getKind() const { return Kind; }
    llvm::StringRef getText() const { return Text; }

    // to test if the token is of a certain kind
    bool is(TokenKind K) const { return Kind == K; }
    bool isOneOf(TokenKind K1, TokenKind K2) const
    {
        return is(K1) || is(K2);
    }
    template <typename... Ts>
    bool isOneOf(TokenKind K1, TokenKind K2, Ts... Ks)
        const { return is(K1) || isOneOf(K2, Ks...); }
};
```

این متدها، getters و ابزارهایی هستند برای بررسی نوع توکن:

- `getKind()`: نوع توکن را بازمی‌گرداند.
- `getText()`: متن مرتبط با توکن را بازمی‌گرداند.
- `is()` و `isOneOf()`: متدهای کمکی برای بررسی اینکه آیا یک توکن با نوع خاصی مطابقت دارد. این متدها هنگام پارس کردن برای شناسایی الگوهای توکن استفاده می‌شوند.

کلاس Lexer

```
class Lexer
{
    const char *BufferStart; // pointer to the beginning of the input
    const char *BufferPtr;   // pointer to the next unprocessed character

public:
    Lexer(const llvm::StringRef &Buffer)
    {
        BufferStart = Buffer.begin();
        BufferPtr = BufferStart;
    }

    void next(Token &token); // return the next token
    void setBufferPtr(const char* buffer);
    const char* getBuffer(){return BufferPtr;};
```

کلاس Lexer مسئول خواندن کد source و تقسیم آن به توکن‌های جداگانه است. این کلاس ورودی را از طریق `BufferStart` و `BufferPtr` دنبال می‌کند.

- `next()`: این متد Lexer را جلو می‌برد و توکن بعدی از ورودی را در توکن ارسالی قرار می‌دهد.
- `setBufferPtr()`: امکان بازنشانی موقعیت Lexer در داخل بافر ورودی را فراهم می‌کند (برای مثال، برای پیش‌خوانی یا بازگشت به عقب).
- `getBuffer()`: موقعیت فعلی در بافر را بازمی‌گرداند.

```
private:
    void formToken(Token &Result, const char *TokEnd, Token::TokenKind Kind);
};
```

این متد کمکی یک توکن با نوع داده شده (Kind) و اشاره‌گری به انتهای توکن (TokEnd) می‌سازد و متن و نوع آن را به‌طور مناسب تنظیم می‌کند. این متد به‌طور داخلی توسط Lexer زمانی که توکن‌ها را در ورودی شناسایی می‌کند، استفاده می‌شود.

Lexer.cpp

کلاس Lexer ورودی را کاراکتر به کاراکتر پردازش کرده و توکن‌هایی که اجزای مختلف کد را نشان می‌دهند، تولید می‌کند. این توکن‌ها شامل شناسه‌ها، کلمات کلیدی، اعداد، عملگرها و نشانه‌ها هستند. متد next() اصلی‌ترین بخش تحلیلگر واژگانی است که این فرآیند را انجام می‌دهد و با کمک توابعی مانند charinfo::isWhitespace() و isLetter() کاراکترها را طبقه‌بندی می‌کند.

```
namespace charinfo
{
    LLVM_READNONE inline bool isWhitespace(char c) { ... }
    LLVM_READNONE inline bool isDigit(char c) { ... }
    LLVM_READNONE inline bool isLetter(char c) { ... }
    LLVM_READNONE inline bool isSpecialCharacter(char c) { ... }
}
```

charinfo توابع کمکی برای دسته‌بندی کاراکترها فراهم می‌کند:

- isWhitespace(): بررسی می‌کند که آیا کاراکتر فضای خالی، تب، خط جدید یا سایر کاراکترهای فضای خالی است.
- isDigit(): بررسی می‌کند که آیا کاراکتر یک عدد (0-9) است.
- isLetter(): بررسی می‌کند که آیا کاراکتر یک حرف الفبا (a-z یا A-Z) است.
- isSpecialCharacter(): بررسی می‌کند که آیا کاراکتر، کاراکتر خاصی مانند عملگرها و علائم نگارشی است (مثل =، +، - و غیره).

```

void Lexer::next(Token &token)
{
    while (*BufferPtr && charinfo::isWhitespace(*BufferPtr)) {
        ++BufferPtr; // Skip over any whitespace characters.
    }

    if (!*BufferPtr) {
        token.Kind = Token::eoi; // End of input
        return;
    }
}

```

متد next() ورودی را پردازش کرده و توکن بعدی را تولید می‌کند.

Lexer از روی هر کاراکتر فضای خالی در ابتدای بافر ورودی عبور می‌کند.

اگر Lexer به انتهای بافر ورودی برسد، نوع توکن را به Token::eoi (پایان ورودی) تنظیم کرده و بازمی‌گردد.

شناسایی شناسه‌ها و کلمات کلیدی

```

if (charinfo::isLetter(*BufferPtr)) {
    const char *end = BufferPtr + 1;
    while (charinfo::isLetter(*end) || charinfo::isDigit(*end))
        ++end;

    llvm::StringRef Name(BufferPtr, end - BufferPtr);
    Token::TokenKind kind;
    // Checking for keywords
    if (Name == "int")
        kind = Token::KW_int;
    // other keywords...

    // Default to identifier if not a keyword
    else
        kind = Token::ident;

    formToken(token, end, kind);
    return;
}

```

اگر Lexer به یک حرف برخورد کند، شروع به جمع‌آوری کاراکترها برای یک شناسه (یا یک کلمه کلیدی) می‌کند. از یک حلقه برای جمع‌آوری حروف یا اعداد بعدی استفاده می‌کند.

پس از جمع‌آوری کاراکترها، Lexer بررسی می‌کند که آیا رشته یک کلمه کلیدی شناخته‌شده است (مثل `int`، `bool`). اگر نه، به طور پیش‌فرض به `Token::ident` (شناسه) تنظیم می‌شود.

شناسایی اعداد

```
else if (charinfo::isDigit(*BufferPtr)) {  
    const char *end = BufferPtr + 1;  
    while (charinfo::isDigit(*end))  
        ++end;  
  
    formToken(token, end, Token::number);  
    return;  
}
```

اگر Lexer به یک رقم برخورد کند، تمام ارقام بعدی را جمع‌آوری کرده و یک توکن عددی تشکیل می‌دهد.

شناسایی علائم خاص (عملگرها و نشانه‌ها)

```
else if (charinfo::isSpecialCharacter(*BufferPtr)) {
    const char *endWithOneLetter = BufferPtr + 1;
    const char *endWithTwoLetter = BufferPtr + 2;
    llvm::StringRef NameWithOneLetter(BufferPtr, endWithOneLetter - BufferPtr);
    llvm::StringRef NameWithTwoLetter(BufferPtr, endWithTwoLetter - BufferPtr);

    // Check for two-character tokens (e.g., '==', '!=')
    if (NameWithTwoLetter == "==") {
        kind = Token::eq;
        end = endWithTwoLetter;
    }
    // Check for one-character tokens (e.g., '=')
    else if (NameWithOneLetter == "=") {
        kind = Token::assign;
        end = endWithOneLetter;
    }
    // Other token checks...

    formToken(token, end, kind);
    return;
}
```

توکن‌های دو کاراکتری مانند == یا != ابتدا بررسی می‌شوند. اگر یافت نشوند، توکن‌های یک کاراکتری مانند = پردازش می‌شوند.

اگر Lexer به یک کاراکتر ناشناخته (که نه حرف، نه رقم و نه کاراکتر خاص باشد) برخورد کند، یک توکن ناشناخته ایجاد می‌کند.

```
void Lexer::setBufferPtr(const char *buffer) {
    BufferPtr = buffer;
}
```

این متد امکان تنظیم اشاره‌گر بافر Lexer به یک موقعیت خاص را فراهم می‌کند. این کار زمانی مفید است که بخواهید موقعیت Lexer را در ورودی بازنشانی یا تنظیم کنید.


```
void Lexer::formToken(Token &Tok, const char *TokEnd, Token::TokenKind Kind)
{
    Tok.Kind = Kind;
    Tok.Text = llvm::StringRef(BufferPtr, TokEnd - BufferPtr);
    BufferPtr = TokEnd;
}
```

متد () formToken یک توکن با نوع و متن مشخص ایجاد می‌کند و اشاره‌گر ورودی را به مکان جدید منتقل می‌کند.

AST.h

این فایل با نام **AST.h** ساختار **Abstract Syntax Tree** را برای یک زبان برنامه نویسی خاص تعریف می‌کند. AST یک نمایش درختی از کد منبع است که گره‌های آن شامل انواع مختلف دستورات و عبارات در یک برنامه می‌باشند.

خلاصه اجزا

- **کلاس‌های AST:** این کلاس‌ها شامل انواع مختلف گره‌های AST مانند عبارات (**Expr**)، برنامه‌ها (**Program**)، عملیات‌های باینری (**BinaryOp**)، تخصیص‌ها (**Assignment**)، و ساختارهای کنترلی مانند شرط‌ها (**IfStmt**) و حلقه‌ها (**ForStmt** و **WhileStmt**) هستند.
- **الگوریتم Visitor:** یک الگوی طراحی است که به کلاس‌ها اجازه می‌دهد تا به بازدیدکننده‌های مختلف اجازه دهند گره‌های مختلف درخت را پیمایش و پردازش کنند.

به عنوان مثال، کلاس BinaryOp

این کلاس نشان‌دهنده یک عملیات دودویی (Binary Operation) در AST است. عملیات دودویی عملیاتی است که دو عملوند (چپ و راست) دارد، مانند جمع، تفریق، ضرب و تقسیم.

```
class BinaryOp : public Expr
```

```

{
public:
    enum Operator
    {
        Plus,      // +
        Minus,     // -
        Mul,       // *
        Div,       // /
        Mod,       // %
        Exp        // ^
    };

private:
    Expr *Left;    // عملوند سمت چپ
    Expr *Right;   // عملوند سمت راست
    Operator Op;   // (بالا enum عملگر دودویی (یکی از موارد)

public:
    BinaryOp(Operator Op, Expr *L, Expr *R) : Op(Op), Left(L), Right(R) {}

    Expr *getLeft() { return Left; } // گرفتن عملوند چپ
    Expr *getRight() { return Right; } // گرفتن عملوند راست
    Operator getOperator() { return Op; } // گرفتن نوع عملگر

    virtual void accept(ASTVisitor &V) override
    {
        V.visit(*this);
    }
};

```

- این کلاس برای بیان عملیات دودویی بین دو عبارت (**Right** و **Left**) استفاده می‌شود.
- عملگرها مانند +، -، *، / و غیره در قالب یک **enum** تعریف شده‌اند.
- این کلاس یک سازنده دارد که عملوند چپ و راست و عملگر را به عنوان ورودی می‌گیرد و آن‌ها را تنظیم می‌کند.
- متد **accept** برای پشتیبانی از الگوی Visitor است، به طوری که یک بازدیدکننده می‌تواند این گره از درخت را پردازش کند.

مثال:

فرض کنید شما یک عبارت ریاضی مثل $5 + 3$ دارید. این عبارت در AST به صورت یک گره `BinaryOp` نشان داده می‌شود که:

- عملگر آن **Plus** است.
- عملوند چپ آن عدد 3 و عملوند راست آن عدد 5 است.

کلاس IfStmt

این کلاس نشان‌دهنده یک **دستور شرطی if** در AST است. دستور شرطی if برای بررسی یک شرط و اجرای مجموعه ای از دستورات در صورت صحیح بودن شرط استفاده می‌شود. همچنین این کلاس از else و elif نیز پشتیبانی می‌کند.

```
class IfStmt : public Program
{
using BodyVector = llvm::SmallVector<AST *>;
using elifVector = llvm::SmallVector<elifStmt *>;

private:
    BodyVector ifStmts;    // دستورات داخل بخش if
    elifVector elifStmts; // دستورات elif
    BodyVector elseStmts;  // دستورات بخش else
    Logic *Cond;           // شرط if

public:
    IfStmt(Logic *Cond, llvm::SmallVector<AST *> ifStmts,
            llvm::SmallVector<AST *> elseStmts, llvm::SmallVector<elifStmt *>
            elifStmts)
        : Cond(Cond), ifStmts(ifStmts), elseStmts(elseStmts),
          elifStmts(elifStmts) {}

    Logic *getCond() { return Cond; } // دریافت شرط

    BodyVector::const_iterator begin() { return ifStmts.begin(); } // دستورات if
    BodyVector::const_iterator end() { return ifStmts.end(); }

    BodyVector::const_iterator beginElse() { return elseStmts.begin(); } // دستورات else
    BodyVector::const_iterator endElse() { return elseStmts.end(); }

    elifVector::const_iterator beginElif() { return elifStmts.begin(); } // دستورات elif
    elifVector::const_iterator endElif() { return elifStmts.end(); }
```

```
virtual void accept(ASTVisitor &V) override
{
    V.visit(*this);
}
};
```

- این کلاس برای نمایش یک دستور **if** با گزینه‌های **elif** و **else** در AST طراحی شده است.
- شرط **if** به وسیله یک شیء از نوع **Logic** (شرط منطقی) تعریف می‌شود.
- مجموعه دستورات **if**، **elif** و **else** به صورت لیست‌هایی از گره‌های AST (توسط SmallVector ها) ذخیره می‌شوند.
- متدهای مختلفی برای دسترسی به شرط **if**، دستورات **else**، **if** و **elif** وجود دارد.

فرض کنید یک دستور زیر را داریم:

```
if (x > 10) {
    print("x is greater than 10");
} else {
    print("x is 10 or less");
}
```

این دستور در AST به صورت یک گره **IfStmt** مدل می‌شود که:

- شرط آن **x > 10** است (که یک گره **Comparison** خواهد بود).
- دستورات **if** شامل چاپ "x is greater than 10" هستند.
- بخش **else** شامل چاپ "x is 10 or less" خواهد بود.

Parser.h

Parser.cpp

همونطور که میدونید قراره توی این پروژه برای یک زبان فرضی با دستورات مشخص یک کامپایلر طراحی کنیم. به زبان دیگه باید بتونیم تشخیص بدیم یک عبارت ورودی جزو زبان تعریف شده هست یا نه و همینطور ارور داره این عبارت یا خیر. برای این کار باید گام به گام پیش بریم، قدم اول اینه که عبارت رو به توکن‌های سازنده اش

تجزیه کنیم و بررسی کنیم هر توکن valid هست یا نه که در بخش لکسر مفصل توضیح داده شده. قدم دوم اینه که اگه توکن‌ها valid بودن بیایم بررسی کنیم که آیا ترتیب قرار گرفتن این توکن‌ها درست هست یا نه یا دقیق‌تر بگم، این عبارت ورودی مجموعه ای از دستورات تعریف شده‌ی زبان هست یا خیر که پیاده‌سازی این بخش در پارسر انجام میشه. ما نیاز داریم دستورات عبارت ورودی رو به صورت یک لیست داشته باشیم و برای هر دستوری که در صورت پروژه تعریف شده یک تابع parser بنویسیم تا بتونه چک کنه با زبان مطابقت داره یا نه. برای درک بهتر به مثال توجه کنید. در لینک داده شده برای فاز اول یک تابع اصلی parseProgram داریم که در ادامه توضیح میدم، همچنین به ازای هر دستور یک تابع پارسر تعریف شده.

تابع parseProgram پارسر اولیه ما هست، این شکلی کار میکنه که لیست دستورات ورودی رو داره و با توجه به توکن اول هر دستور، تشخیص میده که دستور چیه و پارسر مختص اون دستور رو صدا میزنه تا بررسی شه دستور درسته یا نه.

```
11 Program *Parser::parseProgram()
12 {
13     llvm::SmallVector<AST *> data;
14
15     while (!Tok.is(Token::eoi))
16     {
17         switch (Tok.getKind())
18         {
19             case Token::KW_int: {
20                 DeclarationInt *d;
21                 d = parseIntDec();
22                 if (d)
23                     data.push_back(d);
24                 else
25                     goto _error;
26
27                 break;
28             }
```

این تصویر بخشی از تابع parseProgram رو نشون میده و لیست دستورات هم data نام داره. برای مثال به تابع مختص دستور پرینت توجه کنید. سینتکس دستور پرینت که در صورت پروژه تعریف شده: print(var);

که خود var نام یک متغیر هست. قراره توکن به توکن جلو بریم و چک کنیم ورودی ما خارج از تعریف زبان نباشه.

```

867     PrintStmt *Parser::parsePrint()
868     {
869         llvm::StringRef Var;
870         if (expect(Token::KW_print)){
871             goto _error;
872         }
873         advance();
874         if (expect(Token::l_paren)){
875             goto _error;
876         }
877         advance();
878         if (expect(Token::ident)){
879             goto _error;
880         }
881         Var = Tok.getText();
882         advance();
883         if (expect(Token::r_paren)){
884             goto _error;
885         }
886         advance();
887         if (expect(Token::semicolon)){
888             goto _error;
889         }
890         return new PrintStmt(Var);
891
892     _error:
893         while (Tok.getKind() != Token::eoi)
894             advance();
895         return nullptr;

```

همونطور که میبینیم هر توکن بررسی میشه، با کمک تابع advance به توکن بعد میریم و اگه در این حین چیزی سر جای خودش نبود ارور مناسب رو نمایش میدیم. از خط 870 ابتدا توکن کلمه پرینت چک میشه بعد پرانتز باز، سپس نام متغیر بعد پرانتز بسته و در انتها هم سمیکالن. حالا اگه تمام این مراحل رو رد کردیم و عبارت ورودی با موفقیت به انتها رسید یک آبجکت از نوع printStmt میسازه و ورودی(های) لازم رو بهش میده که تو این مثال فقط Var هست. به همین ترتیب باید برای بقیه دستورات هم تابع پارسرشون رو طراحی کنید. همچنین در فایل parser.h هم قراره توابع مورد نیاز مثل expect، advance و... رو تعریف کنید و امضای توابع که در parser.cpp قراره بسازین رو ذکر کنین.

Sema.h

فایل Sema.h هدر اصلی مرتبط با فایل Sema.cpp است. این فایل یک کلاس ساده به نام Sema را تعریف می‌کند که نقطه شروع تحلیل معنایی برنامه است.

Sema.cpp

فایل Sema.cpp مسئول تحلیل معنایی (Semantic Analysis) است. هدف این بخش، بررسی قوانین معنایی زبان برنامه نویسی است؛ مثل بررسی اینکه آیا متغیرها به درستی تعریف شده‌اند و یا نوع داده‌ها در عملیات ریاضی و منطقی به درستی استفاده شده‌اند.

وظیفه تحلیل معنایی

تحلیل معنایی به طور کلی وظیفه دارد که:

- بررسی کند آیا تمام متغیرها قبل از استفاده تعریف شده‌اند.
- نوع داده‌ی متغیرها را در طول عملیات بررسی کند.
- از معتبر بودن عملیات‌هایی مانند تقسیم بر صفر جلوگیری کند.
- مطمئن شود که مقایسه‌ها و عملیات منطقی بر متغیرهای درست اجرا می‌شوند (مثلاً نمی‌توان عملیات ریاضی را بر متغیرهای بولی انجام داد).

نحوه کار فایل Sema

1. ابتدا درخت نحوی انتزاعی (AST) که از مرحله Parser تولید شده است، به Sema داده می‌شود.
2. شیء InputCheck با استفاده از متد accept شروع به پیمایش درخت می‌کند.
3. در هر گره از درخت (مثل گره‌های متغیرها، عملگرها، و دستورات کنترلی)، قوانین معنایی بررسی می‌شوند. به عنوان مثال:
 - آیا متغیری قبل از استفاده تعریف شده است؟
 - آیا عملیات‌های ریاضی بر روی نوع داده‌ی درست انجام شده‌اند؟
 - آیا تقسیم بر صفر انجام نشده است؟
4. در صورتی که در هر مرحله خطایی وجود داشته باشد، پیام خطا از طریق `llvm::errs` چاپ می‌شود و پرچم `HasError` به true تغییر می‌کند.
5. در نهایت نتیجه بررسی به صورت true یا false بازگردانده می‌شود که نشان دهنده وجود یا عدم وجود خطا در کد است.

خطاهای رایج

- تعریف دوباره متغیر: اگر متغیری با نامی که قبلاً تعریف شده دوباره تعریف شود، خطا گزارش می‌شود.
- تعریف نشده بودن متغیر: اگر متغیری قبل از استفاده تعریف نشده باشد، خطا رخ می‌دهد.
- استفاده نادرست از نوع داده‌ها: اگر عملیات ریاضی یا منطقی بر روی نوع داده‌ی نادرستی (مثل بولی به جای عدد صحیح) انجام شود، خطا رخ می‌دهد.
- تقسیم بر صفر: اگر تلاش برای تقسیم بر صفر انجام شود، خطا گزارش می‌شود.

CodeGen.cpp

این فایل بخشی از LLVM backend است.

تابع‌های visit در فایل کدجن دوباره override می‌شوند (هم در کدجن و هم در سمنتیک override می‌شوند). در واقع با accept کردن نودها در این فایل، تابع‌های visit در کدجن فراخوانی می‌شوند.

```
namespace
ns{
    class ToIRVisitor : public ASTVisitor
    {
        Module *M;
        IRBuilder<> Builder;
        Type *VoidTy;
        Type *Int1Ty;
        Type *Int32Ty;
        Type *Int8PtrTy;
        Type *Int8PtrPtrTy;
        Constant *Int32Zero;
        Constant *Int32One;
        Constant *Int1False;
        Constant *Int1True;

        Value *V;
        StringMap<AllocaInst *> nameMapInt;
        StringMap<AllocaInst *> nameMapBool;

        FunctionType *PrintIntFnTy;
        Function *PrintIntFn;

        FunctionType *PrintBoolFnTy;
        Function *PrintBoolFn;
```

کلاس ToIRVisitor تعریف شده است که وظیفه آن پیمایش درخت AST و تولید کد IR assembly معادل آن است. Builder شی ای از IRBuilder است که متدهایی برای ایجاد و درج دستورات LLVM IR ارائه می‌دهد. این

شیء محل درج فعلی دستورات را نگه می‌دارد. متغیر `V` مقدار محاسبه‌شده فعلی است که از طریق پیمایش درخت به‌روزرسانی می‌شود (به عنوان مثال تابع `visit` برای `BinaryOp` را ببینید). `NamemapInt` و `NameMapBool` یک سری نام متغیر را به خانه‌های حافظه `map` می‌کند که برای چک کردن اینکه کدام متغیر از قبل تعریف شده، مفید است.

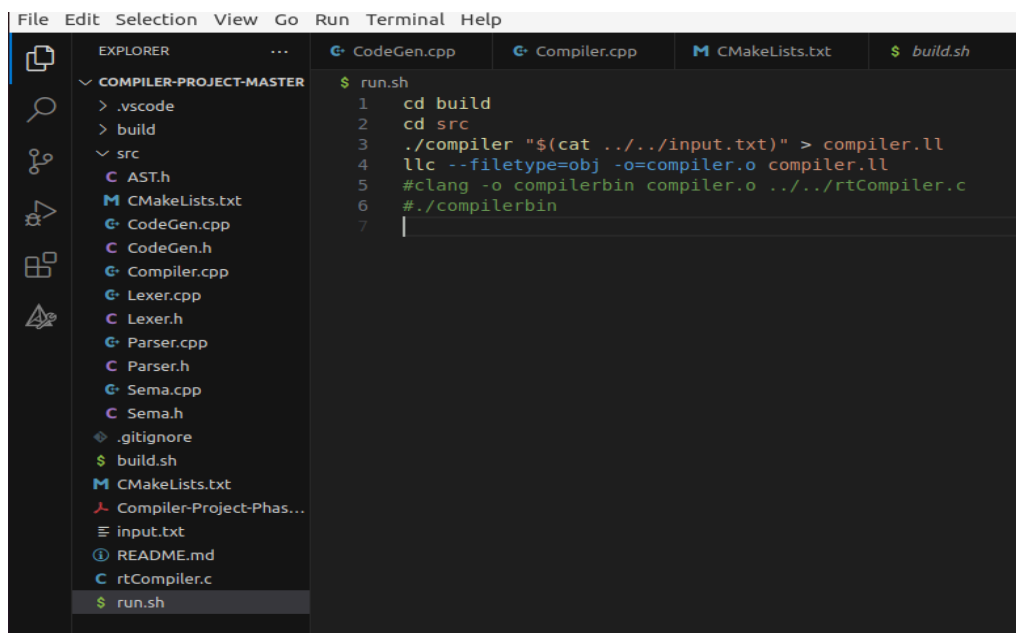
CodeGen.h

در این فایل، کلاس `CodeGen` تعریف می‌شود که شامل تابع `compile` و آرگومان درخت است.

روش تست هر فایل به شکل مجزا

بهرتر است بعد از پیاده‌سازی `Lexer`، `AST` و `Parser` کدتان را تست کنید تا از وجود باگ‌ها هر چه سریع‌تر آگاه شوید. بعد از اطمینان از صحت بخش‌های اولیه، بخش‌های بعدی (`semantic` و `codegen`) را پیاده‌سازی کنید.

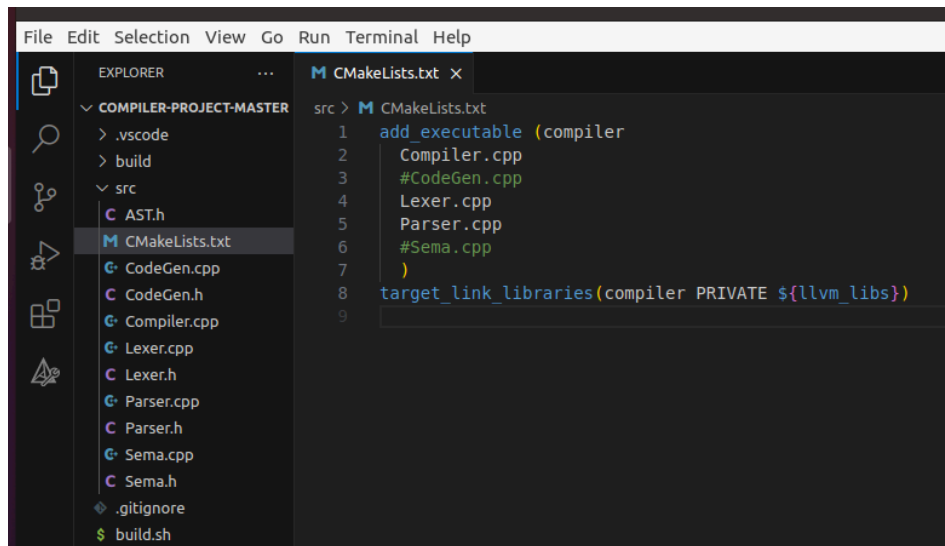
۱. ابتدا خطوط مربوط به `clang` را در فایل `run.sh` کامنت کنید چون این دو خط مربوط به بخش‌های آخر پروژه (`codegen`) است.



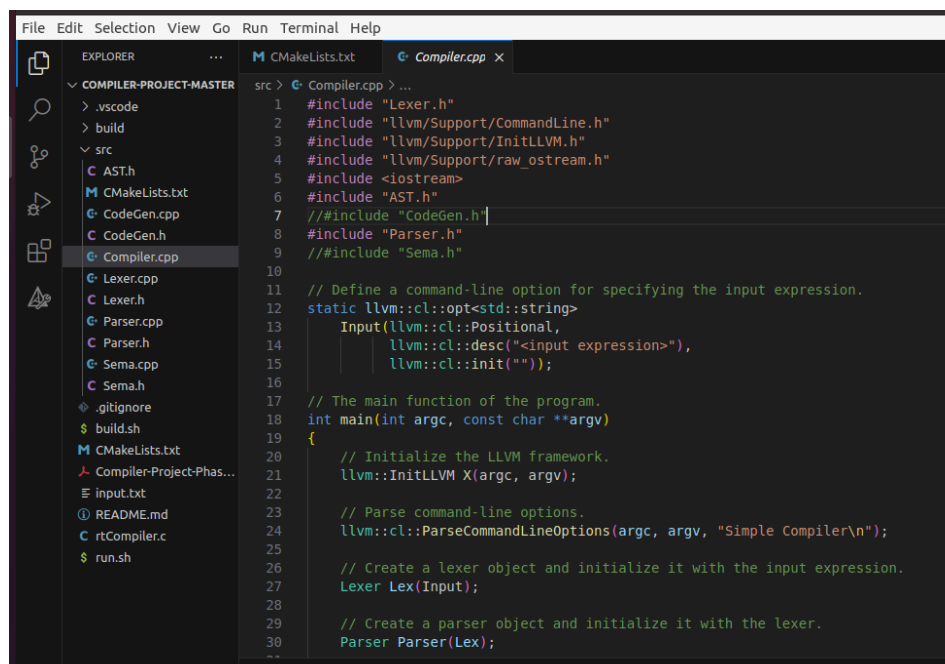
```
File Edit Selection View Go Run Terminal Help
COMPILER-PROJECT-MASTER
  > .vscode
  > build
  > src
    C AST.h
    M CMakeLists.txt
    G CodeGen.cpp
    C CodeGen.h
    G Compiler.cpp
    G Lexer.cpp
    C Lexer.h
    G Parser.cpp
    C Parser.h
    G Sema.cpp
    C Sema.h
    .gitignore
    $ build.sh
    M CMakeLists.txt
    Compiler-Project-Phas...
    input.txt
    README.md
    C rtCompiler.c
    $ run.sh

$ run.sh
1 cd build
2 cd src
3 ./compiler "$(cat ../../input.txt)" > compiler.ll
4 llc --filetype=obj -o=compiler.o compiler.ll
5 #clang -o compilerbin compiler.o ../../rtCompiler.c
6 #./compilerbin
7 |
```

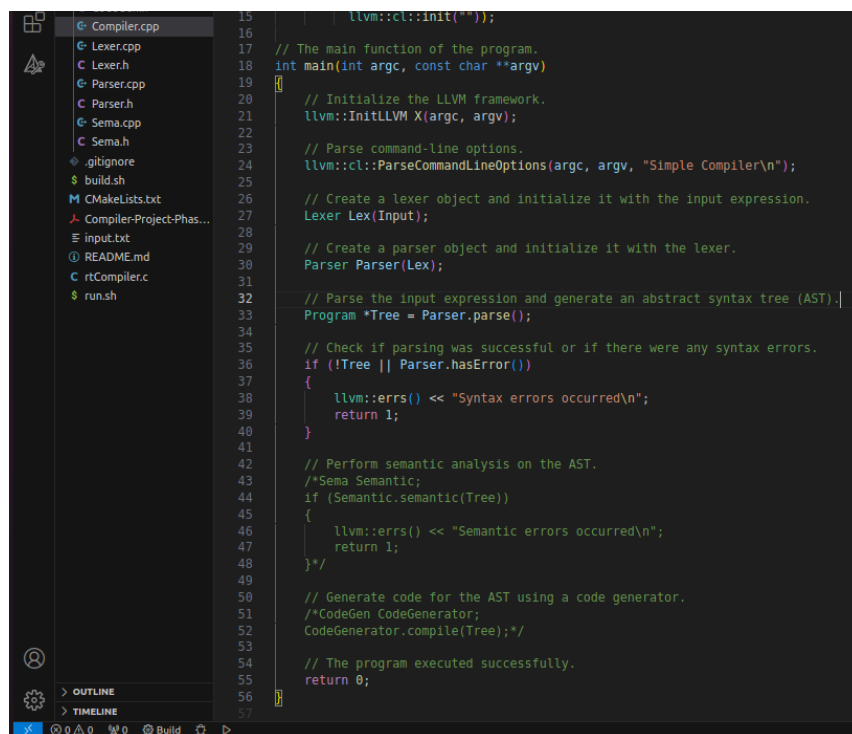
۲. در فایل CMakeLists نام فایل‌های codegen و semantic را کامنت کنید.



۳. در فایل Compiler.cpp یا Main.cpp در بخش include فایل‌های codegen و semantic را کامنت کنید.



۴. همچنین در آخر فایل، کدهای مربوط به فراخوانی semantic و codegen را کامنت کنید.



```
15 | | llvm::cl::init("");
16 |
17 | // The main function of the program.
18 | int main(int argc, const char **argv)
19 | {
20 |     // Initialize the LLVM framework.
21 |     llvm::InitLLVM X(argc, argv);
22 |
23 |     // Parse command-line options.
24 |     llvm::cl::ParseCommandLineOptions(argc, argv, "Simple Compiler\n");
25 |
26 |     // Create a lexer object and initialize it with the input expression.
27 |     Lexer Lex(Input);
28 |
29 |     // Create a parser object and initialize it with the lexer.
30 |     Parser Parser(Lex);
31 |
32 |     // Parse the input expression and generate an abstract syntax tree (AST).
33 |     Program *Tree = Parser.parse();
34 |
35 |     // Check if parsing was successful or if there were any syntax errors.
36 |     if (!Tree || Parser.hasError())
37 |     {
38 |         llvm::errs() << "Syntax errors occurred\n";
39 |         return 1;
40 |     }
41 |
42 |     // Perform semantic analysis on the AST.
43 |     /*Sema Semantic;
44 |     if (Semantic.semantic(Tree))
45 |     {
46 |         llvm::errs() << "Semantic errors occurred\n";
47 |         return 1;
48 |     }*/
49 |
50 |     // Generate code for the AST using a code generator.
51 |     /*CodeGen CodeGenerator;
52 |     CodeGenerator.compile(Tree);*/
53 |
54 |     // The program executed successfully.
55 |     return 0;
56 | }
```

حال می‌توانید با build و run کردن، پارسر خود را دیباگ کنید.

پ.ن: اگر خواستید semantic را هم دیباگ کنید، همین مراحل را انجام دهید فقط دیگر بخش‌های مربوط به semantic را کامنت نکنید. بعد از پیاده‌سازی codegen تمام این تغییرات ذکر شده را به حالت اول برگردانید.

نحوه لاگ گذاشتن در کد برای دیباگ

هر جا خواستید در کد لاگ بگذارید، می‌توانید از llvm::errs استفاده کنید:

```
llvm::errs() << "Semantic errors occurred\n";
```

لینک پروژه‌های گیت‌هاب ترم قبل

**پیشنهاد می‌شود صورت پروژه ترم‌های قبل و گرامر آنها را بخوانید تا متوجه شباهت‌ها و تفاوت‌ها شوید. یکی از پروژه‌های زیر را clone بگیرید. همچنین از اجرا شدن آن روی سیستم خود اطمینان حاصل کنید. سپس تغییرات مربوط به پروژه خود را در آن ایجاد کنید.

<https://github.com/RozhanMk/Compiler-Project>

<https://github.com/mnakhjiri/Compiler-Project>

لینک پروژه‌های گیت‌هاب دو ترم قبل

<https://github.com/AliLRS/Compiler-Project>

<https://github.com/Mohammad-Momeni/MAS-Lang>

باتشکر از نویسندگان داکيومنت : روزان ميرزایی، محمد نیک فلاح، مبینا شهبازی، پارسا عصمت لو